

VLSI IMPLEMENTATION OF PIPELINED QUADRATIC FUNCTION

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIRMENTS FOR THE DEGREE OF

Master of Technology
In
VLSI Design and Embedded Systems

By
NARESH KUMAR KOPPALA
Roll No: 207EC202



Department of Electronics & Communication Engineering
National Institute of Technology, Rourkela
2007-2009

VLSI IMPLEMENTATION OF PIPELINED QUADRATIC FUNCTION

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIRMENTS FOR THE DEGREE OF

**Master of Technology
In
VLSI Design and Embedded Systems**

By
NARESH KUMAR KOPPALA
Roll No: 207EC202

Under the Guidance of
PROF.G.S.RATH



**Department of Electronics & Communication Engineering
National Institute of Technology, Rourkela
2007-2009**



National Institute Of Technology Rourkela

C E R T I F I C A T E

This is to certify that the thesis entitled. “**VLSI IMPLEMENTATION OF PIPELINED QUADRATIC FUNCTION**” submitted by Mr. NARESH KUMAR KOPPALA in partial fulfillment of the requirements for the award of Master of Technology Degree in Electronics and Communication Engineering with specialization in “**VLSI Design and Embedded System**” at the National Institute of Technology, Rourkela (Deemed University) is an authentic work carried out by him under my supervision and guidance.

To the best of my knowledge, the matter embodied in the thesis has not been submitted to any other University/Institute for the award of any Degree or Diploma.

Date:

Prof. G.S. RATH

Place:

Dept. of Electronics and Communication Engg.

National Institute of Technology

Rourkela-769008

ACKNOWLEDGEMENTS

This project is by far the most significant accomplishment in my life and it would be impossible without people who supported me and believed in me.

I would like to extend my gratitude and my sincere thanks to my honorable, esteemed supervisor **Prof. G.S. RATH**, Department of Electronics and Communication Engineering.

He is not only a great lecturer with deep vision but also and most importantly a kind person. I sincerely thank for his exemplary guidance and encouragement. His trust and support inspired me in the most important moments of making right decisions and I am glad to work with him.

I want to thank all my teachers **Prof. G.Panda, Prof. K. K. Mahapatra, Prof. S.K. Patra, Prof. S. Meher** for providing a solid background for my studies and research thereafter. They have been great sources of inspiration to me and I thank them from the bottom of my heart.

I would like to thank all my friends and especially my classmates for all the thoughtful and mind stimulating discussions we had, which prompted us to think beyond the obvious. I've enjoyed their companionship so much during my stay at NIT, Rourkela.

I would like to thank all those who made my stay in Rourkela an unforgettable and rewarding experience.

Last but not least I would like to thank my parents, who taught me the value of hard work by their own example. They rendered me enormous support during the whole tenure of my stay in NIT Rourkela.

NARESH KUMAR KOPPALA

CONTENTS

Abstract	i
List of Figures	ii
List of Tables	iii
Abbreviations	iv
Chapter 1 Introduction	1
1.1. Introduction to Digital Arithmetic	2
1.2. Motivation	2
1.3. Thesis Organization	3
Chapter 2 Computer Number Systems	4
2.1. Introduction to Number System	5
2.2. Conventional Radix Number System	5
2.3. Conversion of Radix Numbers	7
2.4. Representation of Signed Numbers	9
2.4.1. Sign-Magnitude	10
2.4.2. Diminished Radix Complement	10
2.4.3. Radix Complement	11
2.5. Signed-Digit Number System	11
2.6. Floating-Point Number Representation	14
2.6.1 Normalization	15
2.6.2 Bias	16
2.7. Conclusion	18
Chapter 3 Multiplication Algorithms	19
3.1. General Principle of Binary Multiplication	20
3.2. 2's Complement Multiplication - Robertson's Algorithm	21
3.3. Booth's Algorithm	22
3.3.1. First step of Booth's Multiplication	22
3.3.2. Second step of Booth's Multiplication	23
3.3.3. Booth's Multiplication Example	24
3.4. Modified Booth's Multiplication (Booth's Ordering)	25
3.5. Conclusion	27

Chapter 4 Implementation of Pipelined Quadratic Function	28
4.1. General Description	29
4.2. Block Diagram	29
4.3. Key Design Features	30
4.4. Pin-out Descriptions	30
4.5. Generic Parameters	31
4.6. Applications	31
Chapter 5 VHDL Coding & Outputs	32
5.1. Robertson's Multiplication	33
5.2. Booth's Multiplication	35
5.3. Booth's Ordering	37
5.4. Pipelined Quadratic Polynomial	40
5.5. Conclusion	46
Chapter 6 Conclusions and Future work	47
6.1. Conclusions	48
6.2. Future work	48
References	49

ABSTRACT

Arithmetic circuits form an important class of circuits in digital systems. In particular, Multiplication is especially relevant since other arithmetic operators such as division or exponentiation, usually utilizes multiplier as building blocks. As the need for faster computation increases, the need for exploring ways to both quickly and accurately performing the multiplication also increases. So, here we go for Booth's ordering than Booth's multiplication. It provides high performance than other multiplication algorithms.

Very often quadratic and cubic functions are used specifically in the field of cryptography, it is quite relevant at present to implement the pipelined quadratic function and therefore, it has been implemented.

LIST OF FIGURES

Fig. No.	Page No.
2.6.1 Floating-Point Representation	17
2.6.2 Double Precision Floating-point Representation	17
4.1 Block Diagram for implementation of pipelined quadratic function	29

LIST OF TABLES

Table No.		Page No.
2.1	Numbers Represented by 4 bits in Different Number Systems	12
3.1	Example for Booth's Multiplication	24
3.2	Bit Pattern for Booth's Ordering	25
3.3	Register pattern for Booth's Ordering	26
3.4	Example for signed multiplication using Booth's Ordering	26
3.5	Example for unsigned multiplication using Booth's Ordering	27
4.1	Pin-out Description of pipelined quadratic polynomial	30
4.2	Generic Parameter Description	31

ABBREVIATIONS USED

VLSI	Very Large Scale Integration
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Scale Integrated Circuit
MSB	Most Significant Bit
LSB	Least Significant Bit

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION TO DIGITAL ARITHMETIC:

Digital arithmetic operations are very important in the design of digital processors and application-specific systems. Arithmetic circuits form an important class of circuits in digital systems.

With the remarkable progress in the very large scale integration (VLSI) circuit technology, many complex circuits, unthinkable yesterday have become easily realizable today. Algorithms that seemed impossible to implement now, have attractive implementation possibilities for the future. This means that not only the conventional computer arithmetic methods, but also the unconventional ones are worth investigation in new designs.

Multiplication is especially relevant since other arithmetic operators, such as division or exponentiation, which they usually utilize multipliers as building blocks. Hardware implementation of arithmetic operations has been oriented typically to use VLSI circuits. Among the arithmetic operations, the multiplication is widely used in applications such as graphics and scientific computation. Therefore, different kind of schemes has been developed. There are several algorithms for the computation of multiplication. Conventional algorithms include: 1) Robertson's algorithm. 2) Booth's algorithm. 3) Modified Booth's algorithm.

In this work, Implementation of pipelined quadratic function is presented. VHDL (*VHSIC Hardware Description Language*) has been used for circuit description, and the Xilinx tools have been used for the synthesis and simulation.

1.2 MOTIVATION:

Floating point performance is a key denominator of performance for several applications including those in scientific, graphics and DSP domains. High speed floating point hardware is a requirement to meet the ever increasing computational demands of these applications. Modern applications comprise several floating point operations including addition, multiplication, and division. In recent FPUs, emphasis has been placed on designing even faster adders and multipliers, with division receiving less attention. So here we

discussed about different multiplication algorithms and pipelined quadratic polynomial implementation.

1.3 THESIS ORGANIZATION:

In *Chapter 2*, details of Computer Number Systems are discussed. In this chapter, Conventional radix number system, Conversion of radix numbers, Representation of signed numbers, Signed-Digit number system and Floating-Point number representation have been mentioned.

In *Chapter 3*, Conventional multiplication algorithms are discussed. Robertson's multiplication, Booth's algorithm and Modified Booth's algorithm have been discussed.

In *Chapter 4*, Architecture for Pipelined Quadratic Function is described. It consists of design of registers, adders, multipliers, delay-circuits.

In *Chapter5*, Coding part, Synthesis Results and a brief discussion on the result have been put forth..

In *Chapter6*, Conclusions and scope of the extended work is discussed.

CHAPTER 2

COMPUTER NUMBER SYSTEMS

2.1 INTRODUCTION TO NUMBER SYSTEM:

As the arithmetic applications grow rapidly, it is important for computer engineers to be well informed of the essentials of computer number systems and arithmetic processes.

Numbers play an important role in computer systems. Numbers are the basis and object of computer operations. The main task of computers is computing, which deals with numbers all the time.

Humans have been familiar with numbers for thousands of years, whereas representing numbers in computer systems is a new issue. A computer can provide only finite digits for a number representation (fixed word length), though a real number may be composed of infinite digits.

Because of the tradeoffs between word length and hardware size, and between propagation delay and accuracy, various types of number representation have been proposed and adopted. In this book, we introduce the Conventional Radix Number System and Signed-Digit Number System, both belonging to the Fixed-Point Number System, as well as the Floating-point Number System are described.

2.2 CONVENTIONAL RADIX NUMBER SYSTEM:

A conventional radix number N can be represented by a string of n digits such as

$$(d_{n-1}d_{n-2}\dots d_1d_0)_r$$

With r being the radix, d_i $0 \leq i \leq n-1$, is a digit and $d_i \in \{0, 1, \dots, r-1\}$. Note that the position of d_i matters, such as 27 is a different number from 72. Such a number system is referred to as a positional weighted system. Actually,

$$\begin{aligned} N &= d_{n-1} \cdot w_{n-1} + d_{n-2} \cdot w_{n-2} + \dots + d_0 \cdot w_0 \\ &= \sum_{i=0}^{n-1} d_i \cdot w_i \end{aligned}$$

With w_i being the weight of position i . If r is fixed, as in the fixed-radix number system in our further discussion, $w_i = r^i$. Hence,

$$N = d_{n-1}.r^{n-1} + d_{n-2}.r^{n-2} + \dots + d_0.r^0 \text{ -----} > \quad (2.1)$$

$$= \sum_{i=0}^{n-1} d_i \bullet r^i \text{ -----} > \quad (2.2)$$

If r is not fixed, the number becomes a variable-radix number.

To include the fraction into a fixed radix number N , let "." be a radix point with the integer part on the left of it and fraction part on the right of it. There are n digits in the integer and k digits in the fraction, such as

$$(d_{n-1} \dots d_0.d_{-1} \dots d_{-k})_r$$

Then

$$N = \sum_{i=-k}^{n-1} d_i \bullet r^i \text{ -----} > \quad (2.3)$$

For example, in the decimal number system, $r = 10$, and $d_i \in \{0, 1, \dots, 9\}$.

$$\begin{aligned} N &= (69.3)_{10} \\ &= d_1.r^1 + d_0.r^0 + d_{-1}.r^{-1} \\ &= 6 \times 10^1 + 9 \times 10^0 + 3 \times 10^{-1} \\ &= 69.3. \end{aligned}$$

In the octal number system, $r=8$, and $d_i \in \{0, 1, \dots, 7\}$.

$$\begin{aligned} N &= (47.2)_8 \\ &= d_1.r^1 + d_0.r^0 + d_{-1}.r^{-1} \\ &= 4 \times 8^1 + 7 \times 8^0 + 2 \times 8^{-1} \\ &= 32 + 7 + 0.25 = 39.25. \end{aligned}$$

In the hexadecimal number system, $r=16$. Capital letters A through F are used to represent the numbers 10 through 15.

$$\begin{aligned}
 N &= (2A.C)_{16} \\
 &= d_1.r^1 + d_0.r^0 + d_{-1}.r^{-1} \\
 &= 2 \times 16^1 + 10 \times 16^0 + 12 \times 16^{-1} \\
 &= 32 + 10 + 0.75 = 42.75.
 \end{aligned}$$

In the binary number system, $r=2$, and $d_i \in \{0,1\}$.

$$\begin{aligned}
 N &= (10.1)_2 \\
 &= d_1.r^1 + d_0.r^0 + d_{-1}.r^{-1} \\
 &= 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} \\
 &= 2 + 0.5 = 2.5.
 \end{aligned}$$

In the string of weighted digits $(d_{n-1} \dots d_0.d_{-1} \dots d_{-k})_r$, d_{n-1} is called the most significant digit (MSD), and d_{-k} the least significant digit (LSD). A binary digit is referred to as a bit, and the above two digits are MSB and LSB, respectively.

The number of bits required to encode a number λ is $\lfloor \log_2 \lambda \rfloor + 1$.

For example, to represent the decimal number 10, the number of bits required is

$$\lfloor \log_2 10 \rfloor + 1 = \lfloor 3.322 \rfloor + 1 = 4.$$

2.3 CONVERSION OF RADIX NUMBERS:

Numbers can be converted from one radix system to another. The conversion is as follows.

Given an integer,

$$(d_{n-1}d_{n-2} \dots d_1d_0)_r,$$

with base r other than 10, such as $r = 2$ in binary, $r = 8$ in octal or $r = 16$ in hexadecimal, according to Equation (2.2), the following equation provides a method to convert it to the corresponding decimal number N_1 .

$$N_1 = d_{n-1}.r^{n-1} + d_{n-2}.r^{n-2} + \dots + d_0.r^0. \quad \text{-----} > \quad (2.4)$$

That is, N_1 can be obtained by performing the multiplication of each given digit, the weight it carries and summing all the products.

In the reversed way, given a decimal number we can obtain the corresponding digits in its binary, octal or hexadecimal representation by division, using r as the divisor equal to 2, 8 or 16, respectively.

Dividing both sides of Equation (2.4) by r , we have on the right-hand side the remainder d_0 and the quotient

$$d_{n-1}.r^{n-2} + d_{n-2}.r^{n-3} + \dots + d_1,$$

since $d_0 < r$ and other terms on the right-hand side are integer times of r . If we divide the above quotient again by r , we will obtain the remainder d_1 , and so forth. After performing the division $n - 1$ time, d_{n-1} will become the quotient. If we divide it by r again, we will have quotient 0, since any $d_i < r$ and the last remainder d_{n-1} . The conversion procedure stops there.

For a radix r fraction number

$$(0.d_1d_2\dots d_k)_r,$$

With $r \neq 10$, the corresponding decimal number N_2 can be obtained by

$$N_2 = d_1.r^{-1} + d_2.r^{-2} + \dots + d_k.r^{-k}. \quad \text{-----} > \quad (2.5)$$

On the other hand, a decimal fraction can be converted to a radix r number such as a binary, octal or hexadecimal number with r being 2, 8 or 16, respectively.

Multiplying both sides of Equation (2.5) by r , we have on the right-hand side

$$d_1 + d_2.r^{-1} + \dots + d_k.r^{-k+1},$$

where d_1 is the integer part and others add up to the fraction part. Multiply the fraction part by r again, we have

$$d_{-2} + d_{-3}.r^{-1} + \dots + d_{-k}.r^{-k+2}$$

where d_{-2} is the integer part. Repeating the multiplication process and retaining the digits in the integer part, we can obtain the radix r number corresponding to the given decimal fraction. If we stop when the fraction part becomes zero, then the conversion completes precisely. Note that the fraction part may never become zero. Then, depending on how many digits are allowed in the radix r number, one can decide the time to stop the multiplication procedure. Infinite digits may be required to represent the given decimal fraction precisely in the radix r number system. With a limited number of digits, the found radix r number is an approximation.

In octal numbers $r = 8$ and digit $0 \leq d_i \leq 7$. In binary numbers $r = 2$ and $8 = 2^3$. 3 bits in binary are necessary and sufficient to represent the value of one digit in octal. For example, $(5)_8 = (101)_2$ and $(7)_8 = (111)_2$.

In hexadecimal numbers $r = 16$ and digit $0 \leq d_i \leq 15$. In binary numbers $r = 2$ and $16 = 2^4$. 4 bits in binary are necessary and sufficient to represent the value of one digit in hexadecimal. For example, $(9)_{16} = (1001)_2$, and $(15)_{16} = (1111)_2$.

2.4 REPRESENTATION OF SIGNED NUMBERS:

Let a conventional radix number A be an n digit signed number with the MSD representing the sign. That is,

$$A = (a_{n-1}a_{n-2} \dots a_1a_0)_r,$$

And the sign digit is decided as follows:

$$a_{n-1} = \begin{cases} 0 & \text{if } A \geq 0, \\ r-1 & \text{if } A < 0. \end{cases}$$

Note that for an integer number, the radix point is on the right of a_0 that is,

$$(a_{n-1}a_{n-2} \dots a_1a_0.),$$

and for a fraction number, the radix point is on the left of a_{n-2} , such as

$$(a_{n-1}.a_{n-2} \dots a_1 a_0)$$

Particularly, when $a_{n-2} \neq 0$, we say that A is a normalized fraction.

There are three representations of a negative number: **(1)** sign-magnitude, **(2)** diminished radix complement, and **(3)** radix complement.

2.4.1 Sign-Magnitude:

$$(r-1)m_{n-2}m_{n-3} \dots m_1 m_0.$$

Some examples of sign-magnitude representation are:

$$r = 2, (1)1010 = -1010_2 = -10_{10}$$

$$r = 10, (9)7602 = -7602_{10}$$

2.4.2 Diminished Radix Complement:

$$(r-1)\bar{m}_{n-2}\bar{m}_{n-3} \dots \bar{m}_1 \bar{m}_0,$$

where

$$\bar{m}_i = (r-1) - m_i, \quad 0 \leq i \leq n-2$$

The diminished radix complement representation is also known as (r-1)'s complement denoted as

$$\bar{A} = r^n - 1 - |A|,$$

Where n is the total number of digits including the sign digit. For example, given r=2, $\bar{A} = 2^n - 1 - |A|$, and we have the 1's complement representation as follows:

$$-1010_2 : (1)0101.$$

Given $r = 10$, $\bar{A} = 10^n - 1 - |A|$, we have the following 9's complement representation.

$$-7602_{10} : (9)2397.$$

2.4.3 Radix Complement:

$$((r-1)\bar{m}_{n-2}\bar{m}_{n-3}\dots\bar{m}_1\bar{m}_0)+1,$$

where

$$\bar{m}_i = (r-1)-m_i, \quad 0 \leq i \leq n-2.$$

The radix complement representation is also called r 's complement, denoted as

$$\bar{A} = r^n - |A|.$$

For example, given $r=2$, $\bar{A} = 2^n - |A|$, and we have the 2's complement representation as follows:

$$-1010_2 : (1)0110.$$

Given $r=10$, $\bar{A} = 10^n - |A|$, and we have the following 10's complement representation:

$$-7602_{10} : (9)2398.$$

2.5 SIGNED-DIGIT NUMBER SYSTEM:

The number systems introduced in the previous sections belong to the conventional radix number system, which is non-redundant, positional and weighted. Each digit d_i has only a positive value, and is less than radix r . That is,

$$0 \leq d_i \leq r-1.$$

Binary bits	Sign-magnitude	1's Complement	2's Complement
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	-0	-7	-8
1001	-1	-6	-7
1010	-2	-5	-6
1011	-3	-4	-5
1100	-4	-3	-4
1101	-5	-2	-3
1110	-6	-1	-2
1111	-7	-0	-1

Table 2.1: Numbers Represented by 4 bits in Different Number Systems

In signed-digit number system, each digit can have either a positive or a negative value and the representation is redundant.

Let a signed-digit number

$$X = (x_{n-1} \dots x_0 . x_{-1} \dots x_{-k})_r.$$

Given radix $r \geq 2$, each digit of a Signed-Digit Number, x_i , has any value of $\{-\alpha, \dots, -1, 0, 1, \dots, \alpha\}$, that is,

$$-\alpha \leq x_i \leq \alpha,$$

where

$$\lceil (r-1)/2 \rceil \leq \alpha \leq r-1$$

Since the signed-digit representation of a number may not be unique, we choose $\alpha = \lfloor r/2 \rfloor$ for the minimum redundancy.

For example, if $r = 4$, $\alpha = \lfloor 4/2 \rfloor = 2$. Then digit $x_i \in \{-2, -1, 0, 1, 2\}$.

To find the value of X , we have

$$X = \sum_{i=-k}^{n-1} x_i \cdot r^i, \quad x_i \in \{-\alpha, \dots, -1, 0, 1, \dots, \alpha\}.$$

For example, given $r = 4$, $X_{SD} = (1 \bar{2} . 2)_4$, we have

$$\begin{aligned} X &= (1 \times 4^1) + (\bar{2} \times 4^0) + (2 \times 4^{-1}) \\ &= 4 + (-2) + (0.5) \\ &= 2.5. \end{aligned}$$

In general, X can be either positive or negative without putting a "sign" in front of it.

Given $r = 2$, $X_{SD} = (00 \bar{1} \bar{1})_2$, $\alpha = \lfloor 2/2 \rfloor = 1$, $x_i \in \{-1, 0, 1\}$.

$$X = \bar{1} \times 2^1 + \bar{1} \times 2^0 = -3.$$

Note that given $X_{SD} = (0 \bar{1} 01)_2$

$$X = \bar{1} \times 2^2 + 1 \times 2 = -3.$$

2.6 FLOATING-POINT NUMBER REPRESENTATION:

All the number representations discussed so far have the radix point in a fixed position, thus they belong to the fixed-point number representation. In computer systems, the radix point is not actually shown, but its existence and position are mutually agreeable by humans and computers.

The position of the radix point determines the number of digits in the integer part and that in the fraction part. Given the total number of digits fixed, the more digits in the integer the bigger number represented. On the other hand, the more digits in the fraction the better precision obtained. A new idea about floating-point number representation is hereby introduced in contrast to the fixed-point number representation. The general format of floating-point representation is

$$F = (M,E) \approx M \times r^E.$$

Here, M represents the mantissa (or significand), and E the exponent. r is the radix as usual.

For example, $(-0.0025, +3)$ means -0.0025×10^3 for $r = 10$ and is equal to 2.5_{10} .

The mantissa M and exponent E are both signed numbers. M is usually a signed fraction and E a signed integer. Many ways exist to represent the sign, the fraction and the integer. All computers had different representations before the IEEE standard was published. Here we focus on the most common cases.

2.6.1 Normalization:

Most often M is a normalized fraction in the sign-magnitude form. The true magnitude of it is $|M| = (0.m_{-1}m_{-2}..m_{-k})$. that is, $1 \leq |M| < 2$. By normalized, we mean that the MSD of the mantissa equals non-zero, that is, $m_{-1} \neq 0$. If it is zero, the floating-point number is un-normalized.

$$F = (-0.0025, +3)$$

$$= (-0.25, +1)$$

indicates how the normalization procedure can be conducted. Note that a normalized mantissa has its absolute value limited by

$$(1/r) \leq |M| < 1,$$

Where $1/r$ is the weight carried by the MSD in the mantissa. In other words, if $(1/r) \leq |M| < 1$, the floating-point number is normalized. For the binary case with k bits in the mantissa,

$$\frac{1}{2} \leq |M| \leq 1-2^{-k}. \quad \text{-----}> \quad 2.6$$

The reason for normalization is to fully utilize the available bits. Allowing too many leading 0s in the fraction may cause unnecessary truncation of the lower order bits in the mantissa. The bit positions occupied by those leading 0s are wasted, and the accuracy of the number representation is degraded.

For normalization, just perform left shift on the un-normalized mantissa until the leading 0s are all shifted out and the first nonzero digit reaches the most significant position. In the meantime, adjust the exponent accordingly. Left shifting the mantissa for k positions will enlarge the represented number r^k times and deducting k from the exponent can reduce the number r^k times and keep the represented number unchanged.

2.6.2 Bias:

Suppose

$$E = (a_{q-1}a_{q-2}\dots a_0)_2$$

where q is the number of bits. Then,

$$-2^{q-1} \leq E \leq 2^{q-1}-1. \quad \text{-----} > \quad 2.7$$

For example, given $q=4$,

$$-2^3 \leq E \leq 2^3-1.$$

That is,

$$-8 \leq E \leq +7,$$

and

$$(1000)_2 \leq E \leq (0111)_2.$$

Choose a constant as a bias which is usually the magnitude of the lowest bound value of unbiased exponent, that is,

$$\text{bias} = 2^{q-1}.$$

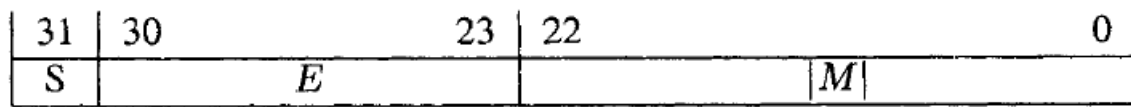
Add the bias to the unbiased exponent; we can have E_{biased} in the following range:

$$0 \leq E_{\text{biased}} \leq 2^q-1.$$

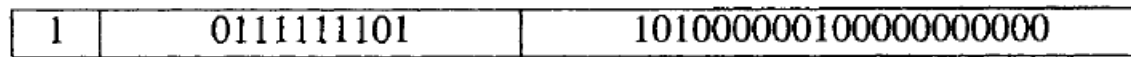
Recall that E_{unbiased} can be either negative or positive, while E_{biased} are all positive now.

The relationship of E_{unbiased} and E_{biased} can be expressed as follows.

$$E_{\text{unbiased}} = E_{\text{biased}} - 2^{q-1}.$$



(a)



(b)

Fig. 2.6.1: Floating-point Representation

Given a number F represented in Figure (2.6.1), one should realize that

$$F = (-1)^S |M| \cdot r^{E-\text{bias}}. \quad \text{-----} > \quad 2.8$$

Here, S has one of the two values, 0 or 1. If $S = 0$, $(-1)^0 = +1$, meaning that the sign of the represented number is positive. If $S = 1$, $(-1)^1 = -1$, meaning that the sign of the represented number is negative.



Fig. 2.6.2: Double Precision Floating-point Representation

The IEEE standard was formulated in 1981. The IEEE single precision format is similar to the one presented in Figure 2.6.1 except in the following protocols. The double precision representation is shown in Figure 2.6.2. What follows is based on the single precision model, while the same principles apply in the double precision case.

Instead of $1/r$ to 1, the range of the mantissa magnitude is 1 to 2 now. Equation (2.8) $F = (-1)^S |M| \cdot r^{E-\text{bias}}$, with $|M| = (0.m_1m_2\dots m_k)$, is now

$$F = (-1)^S (1.m_1m_2\dots m_k) \cdot r^{E-\text{bias}}.$$

The same place used to store $|M|$ is now storing f . The mantissa represented is no longer a pure fraction equal to $\pm|M|$. It has an integer part 1 now, and $\pm 1.f$ is the mantissa which can be viewed as a left shifted normalized fraction. Hence,

$$F = (-1)^S (1.f) \cdot r^{E-\text{bias}}.$$

S is the sign and E is the biased exponent as usual. Another difference here lies in the bias. Rather than $\text{bias} = 2^{q-1}$ in the IEEE standard,

$$\text{bias} = 2^{q-1} - 1 = 127$$

for $q=8$.

While $-128 \leq e_{\text{un-biased}} \leq 127$,

$$1 \leq e_{\text{biased}} \leq 254.$$

2.7 Conclusion:

There are still more number of number systems used in different systems such as cryptography, DSP processes and image compression. However, since the other number systems are not coming in to the picture for this work of implementation of pipelined quadratic polynomial, these are not discussed.

Chapter 3

MULTIPLICATION ALGORITHMS

3.1 GENERAL PRINCIPLE OF BINARY MULTIPLICATION:

A multiplier plays an important role in various digital systems such as computer, process controller and signal processor. Consider two unsigned binary numbers X and Y. The basic algorithm is similar to the one used in multiplying the numbers on pencil and paper. The main operations involved are shift and add. Recall that the 'pencil-and-paper' algorithm is inefficient as in that, each product term (obtained by multiplying each bit of the multiplier to the multiplicand) has to be saved till all such product terms are obtained. In machine implementations, it is desirable to add all such product terms to form the partial product. Also, instead of shifting the product terms to the left, the partial product is shifted to the right before the addition takes place. In other words, if P_i is the partial product after i steps and if Y is the multiplicand and X is the multiplier, then

$$P_i \leftarrow P_i + X_j * Y$$

And

$$P_{i+1} \leftarrow P_i * 2^{-1}$$

and the process repeats.

Note that the multiplication of signed magnitude numbers requires a straight forward extension of the unsigned case. The magnitude part of the product can be computed just as in the unsigned magnitude case. The sign p_0 of the product P is computed from the signs of X and Y as

$$P_0 \leftarrow x_0 \oplus y_0$$

3.2 2'S COMPLEMENT MULTIPLICATION - ROBERTSON'S ALGORITHM:

Consider the case that we want to multiply two 8 bit numbers $X = x_0x_1:::x_7$ and $Y = y_0y_1:::y_7$. Depending on the sign of the two operands X and Y , there are 4 cases to be considered:

$x_0 = y_0 = 0$, that is, both X and Y are positive. Hence, multiplication of these numbers is similar to the multiplication of unsigned numbers. In other words, the product P is computed in a series of add-and-shift steps of the form

$$P_i \leftarrow P_i + X_j * Y$$

$$P_{i+1} \leftarrow P_i * 2^{-1}$$

Note that all partial products are non-negative. Hence, leading 0s are introduced during right shift of the partial product.

$x_0 = 1; y_0 = 0$, that is, X is negative and Y is positive. In this case, the partial product is always positive (till the sign bit x_0 is used). In the final step, a subtraction is performed. That is,

$$P \leftarrow P - Y$$

$x_0 = 0; y_0 = 1$, that is, X is positive and Y is negative. In this case, the partial product is positive and hence leading 0s are shifted into the partial product until the _rst 1 in X is encountered. Multiplication of Y by this 1, and addition to the result causes the partial product to be negative, from which point on leading 1s are shifted in (rather than 0s).

$x_0 = 1$; $y_0 = 1$, that is, both X and Y are negative. Once again, leading 1s are shifted into the partial product whenever the partial product is negative. Also, since X is negative, the correction step (subtraction as the last step) is also performed.

Recall the difference in the correction steps between multiplication of two integers and two fractions. In the case of two integers, the correction step involves subtraction and shift right. Whereas, in the case of fractions, the correction step involves subtraction and setting $Q(7) \leftarrow 0$.

3.3 BOOTH'S ALGORITHM:

Requires that we can do an addition or a subtraction each iteration (not always an addition). When doing multiplication, strings of 0s in the multiplier require only shifting (no addition). When doing multiplication, strings of 1s in the multiplier require an operation only at each end. We need to add or subtract only at positions in the multiplier where there is a transition from a 0 to a 1, or from a 1 to a 0.

3.3.1 First step of Booth's Multiplication:

Look at rightmost 2 bits of multiplier:

00 or 11: do nothing

01: Marks the end of a string of 1s

add multiplicand to partial product (running sum)

10: Marks the beginning of a string of 1s

Subtract multiplicand from partial product.

Initially pretend there is a 0 past the rightmost bit of the multiplier.

3.3.2 Second step of Booth's Multiplication:

The second step of the algorithm is the same as before: shift the product/multiplier right one bit.

Arithmetic shift needed for signed arithmetic.

The bit shifted off the right is saved and used by the next step 1 (which looks at 2 bits).

Booth's algorithm results in reduction in the number of add/subtract steps needed (as compared to the Robertson's algorithm) if the multiplier contains runs (or sequences) of 1s or 0s. The worst case scenario occurs in booth's algorithm if $X = 010101::01$, where there are $n/2$ isolated 1s, which forces $n/2$ subtractions and $n/2$ additions. This is worse than the standard multiplication algorithm

- Which contains only $n/2$ additions.

The basic booth's algorithm can be improved by detecting an isolated 1 in the multiplier and just performing addition at the corresponding point in the multiplication. Similarly, an isolated 0 corresponds to a subtraction. This is called as modified booth's algorithm, which always requires fewer addition/subtractions as compared to other multiplication algorithms.

3.3.3 Booth's Multiplication Example:

29*14(=406)

Multiplicand: 011101

-Multiplicand: 100011

Multiplier: 001110

	0	0	0	0	0	0	0	0	1	1	1	0	0	Starting Product
1	0	0	0	0	0	0	0	0	0	1	1	1	0	In prev. line 00 => Shift right
+	1	0	0	0	1	1	0	0	0	0	0	0	0	In prev. line 10 => sub & shift
2	1	0	0	0	1	1	0	0	0	1	1	1	0	
	1	1	0	0	0	1	1	0	0	0	1	1	1	
3	1	1	1	0	0	0	1	1	0	0	0	1	1	In prev. line 11 => shift right
4	1	1	1	1	0	0	0	1	1	0	0	0	1	In prev. line 11 => shift right
	0	1	1	1	0	1	0	0	0	0	0	0	0	In prev. line 01 => add & shift
5	0	1	1	0	0	1	0	1	1	0	0	0	0	
	0	0	1	1	0	0	1	0	1	1	0	0	0	
6	0	0	0	1	1	0	0	1	0	1	1	0	0	In prev. line 00 => Shift right

Table 3.1: Example for Booth's Multiplication

Solution: 000110010110

3.4 MODIFIED BOOTH'S MULTIPLICATION (BOOTH'S ORDERING):

(A) For sign and 2's complement method:

Let $X = \{x_{n-1}, x_{n-2}, \dots, x_0\}$ and $Y = \{y_{n-1}, y_{n-2}, \dots, y_0\}$ be the n-digit signed numbers represented in sign and 2's complement form. Let Z be defined as the product of X and Y, i.e. $Z = X * Y$. The $Z = \{z_{2n-2}, z_{2n-3}, \dots, z_0\}$ which will have 2n-1 digits. Let the multiplying factor

$m_0 = (-2y_1 + y_0 + y_{-1})2^0$ with $y_{-1} = 0$ is appended to the number after the binary point and

$m_i = (-2y_{2i+1} + y_{2i} + y_{2i-1})2^i$ for $i = 0, 1, \dots, (n-2)/2$ when n is even.

In this ordering we assume that n is even. If we define the partial accumulated product as $Z_{mi} = \sum_{i=0}^i X * m_i 2^i$ Thus depending on the bit pattern the following operation has to be done for finding the partial product Z_{mi+1}

Bit pattern ($-2y_{2i+1}$ y_{2i} y_{2i-1})			Action to be taken
0	0	0	Shift twice if $i < (n-2)/2$
0	0	1	Add X and then shift twice if $i < (n-2)/2$
0	1	0	Add X and then shift twice if $i < (n-2)/2$
0	1	1	Shift once, then add X and the shift once if $i < (n-2)/2$
1	0	0	Shift once, then subtract X and the shift once if $i < (n-2)/2$
1	0	1	Subtract X and then shift twice if $i < (n-2)/2$
1	1	0	Subtract X and then shift twice if $i < (n-2)/2$
1	1	1	Shift twice if $i < (n-2)/2$

Table 3.2: Bit Pattern for Booth's Ordering

The register patterns for such multiplication will be:

C	Z ₁₄	Z ₁₃	Z ₁₂	Z ₁₁	Z ₁₀	Z ₉	Z ₈	Z ₇	Z ₆	Z ₅	Z ₄	Z ₃	Z ₂	Z ₁	Z ₀
---	-----------------	-----------------	-----------------	-----------------	-----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------

Table 3.3: Register pattern for Booth's Ordering

Example : $X = (117)_{10} = 01110101$ and $Y = -(39) = 11011001$

Mi and action to be taken	0	
010 , add X	0 0 1 1 1 0 1 0 1	
Shift twice	0 0 0 0 1 1 1 0 1	0 1
100, shift once	0 0 0 0 0 1 1 1 0	1 0 1
Sub X	1 1 0 0 1 1 0 0 1	1 0 1
Shift once	1 1 1 0 0 1 1 0 0	1 1 0 1
011, shift once	1 1 1 1 0 0 1 1 0	0 1 1 0 1
Add X	0 0 1 0 1 1 0 1 1	0 1 1 0 1
Shift once	0 0 0 1 0 1 1 0 1	1 0 1 1 0 1
110, sub X	1 1 0 1 1 1 0 0 0	1 0 1 1 0 1
Final shift once	1 1 1 0 1 1 1 0 0	0 1 0 1 1 0 1

Table 3.4: Example for signed multiplication using Booth's Ordering.

Final answer: 1 1 0 1 1 1 0 0 0 1 0 1 1 0 1

(B) For Unsigned integer:

Same method is followed excepting that msb of the multiplier(y) must be 0, otherwise 0's must be appended before actual msb in order to complete the pairing the bits to form m_i

Examlpe : $X = (117)_{10} = 01110101$ and $Y = -(89) = 01011001$

Mi and action to be taken	0	
010 , add X	0 0 1 1 1 0 1 0 1	
Shift twice	0 0 0 0 1 1 1 0 1	0 1
100, shift once	0 0 0 0 0 1 1 1 0	1 0 1
Sub X	1 1 0 0 1 1 0 0 1	1 0 1
Shift once	1 1 1 0 0 1 1 0 0	1 1 0 1
011, shift once	1 1 1 1 0 0 1 1 0	0 1 1 0 1
Add X	0 0 1 0 1 1 0 1 1	0 1 1 0 1
Shift once	0 0 0 1 0 1 1 0 1	1 0 1 1 0 1
001, add X	0 1 0 1 0 0 0 1 0	1 0 1 1 0 1
Final shift once	0 0 1 0 1 0 0 0 1	0 1 0 1 1 0 1

Table 3.5: Example for unsigned multiplication using Booth's Ordering

So the answer is $(28AD)_{16} = 10413$ (N.B. 1 is to be added at the left if the resulted number has the borrow otherwise 0 when the number is shifted to the right for the next operation)

3.5 Conclusion:

Only two methods of multiplication have been discussed in detail, since these methods are used for the implementation of pipelined quadratic function.

Chapter -4

IMPLEMENTATION OF PIPELINED QUADRATIC FUNCTION

4.1 GENERAL DESCRIPTION:

QUADRATIC_FUNC is a fully pipelined quadratic polynomial that computes the relation: $y = ax^2 + bx + c$. On each rising-edge of the clock (when en is high), the coefficients and input x term are sampled at the function inputs. The result has a latency of 3 clock cycles. All inputs to the function are 8-bit signed fractions, with the generic parameter fw specifying the number of fraction bits. For example, setting the parameter $fw = 6$ would mean that the x input (and coefficients) would have the format $[8\ 6]$ with 1 sign bit, 1 integer bit and 6 fraction bits. The position of the binary point in the function output is also determined by the number of fraction bits. In this example with $fw = 6$, the output would be in $[24\ 18]$ format with 1 sign bit, 5 integer bits and 18 fraction bits. If integer arithmetic is required throughout, the generic parameter fw should be set to 0. Note that internally, the function does not perform any truncation or rounding of the intermediate results. This means that there is no loss of precision in output result.

4.2 BLOCK DIAGRAM:

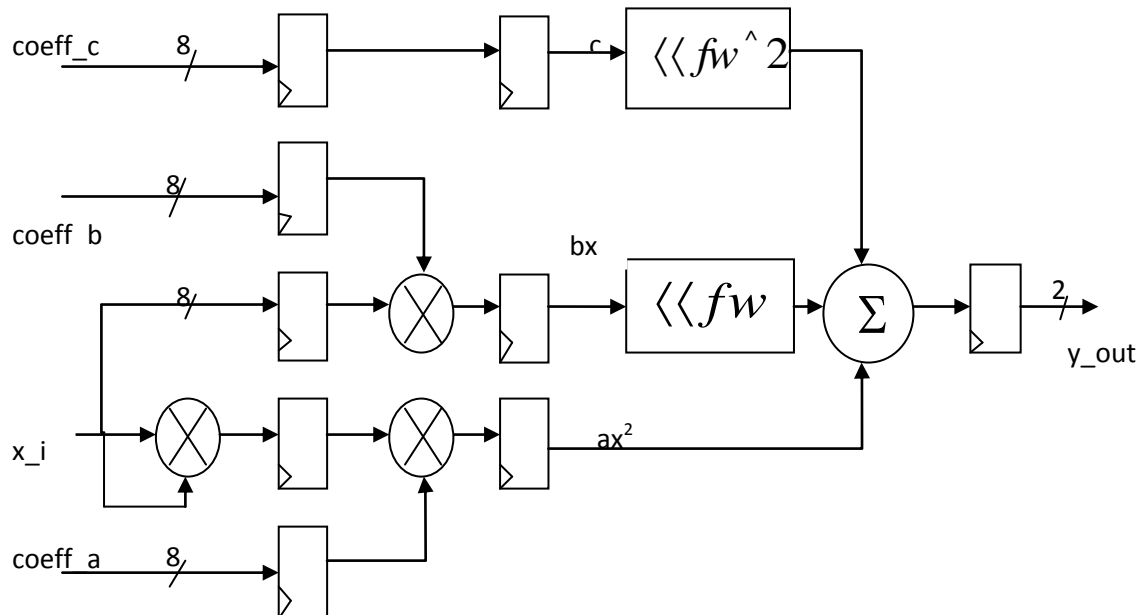


Fig 4.1 Block Diagram for implementation of pipelined quadratic function

4.3 KEY DESIGN FEATURES:

- Computes the relation $y = ax^2 + bx + c$
- Signed 8-bit fixed-point input
- Signed 8-bit fixed-point coefficients
- Signed 24-bit fixed-point output
- Configurable number of fraction bits
- Dynamic coefficients updated every clock-cycle
- No internal loss of precision (no rounding or truncation of intermediate results applied)
- Fully pipelined architecture
- Result has 3 clock-cycle latency

4.4 PIN-OUT DESCRIPTIONS:

Pin name	I/O	Description	Active state
clk	in	Sample clock	rising edge
en	in	Clock-enable	high
coeff_a [7:0]	in	Signed Coefficient a [8 fw] format	data
coeff_b [7:0]	in	Signed Coefficient b [8 fw] format	data
coeff_c [7:0]	in	Signed Coefficient c [8 fw] format	data
x_in [7:0]	in	Signed function input [8 fw] format	data
y_out [23:0]	out	Function output [24 fw*3] format	data

Table 4.1: Pin-out Description of pipelined quadratic polynomial

4.5 GENERIC PARAMETERS:

Generic name	Description	Type	Valid range
fw	Coefficient and x term Fraction width	Integer	[0, 8]

Table 4.2: Generic Parameter Description

4.6 APPLICATIONS:

The polynomial functions are required in many applications. Implementation of pipelined polynomial functions will help in many practical applications such as

- (i) Curve fitting:
 - a) Newton's method of curve fitting requires the functional evaluation of higher degree polynomial.
 - b) Curve-fitting by spline requires the evaluation of (cubic spline) cubic functions which are piecewise continuous.
- (ii) Estimating functions such as SIN, COS, ATAN, Logarithmic functions and Exponential functions.

Circular functions, Logarithmic functions and Exponential functions are evaluated by series method. Therefore evaluation of polynomials is very much required for estimation of above functions.
- (iii) Most of the time, the look-up table is used when the input and output relationship cannot be expressed by bounded functions. This type of nonlinear mapping utilizes the polynomial evaluation.

Chapter -5

VHDL CODING & OUTPUTS

5.1 ROBERTSON'S MULTIPLICATION:

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.STD_LOGIC_ARITH.ALL;

use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity rm is

    Port ( a,b : in  STD_LOGIC_VECTOR (3 downto 0);

          mul : out  STD_LOGIC_VECTOR (7 downto 0));

end rm;

architecture Behavioral of rm is

begin

    process(a,b)

        variable m: std_logic_vector(7 downto 0);

        begin

            m:= "0000"&b;

            for i in 0 to 2 loop

                if m(0)='1' then

                    m(7 downto 4):=m(7 downto 4)+a;

                    m(6 downto 0):=m(7 downto 1);

                else
```

```

m(6 downto 0):=m(7 downto 1);

end if;

end loop;

if b(3)='0' then

if m(0)='1' then

m(7 downto 4):=m(7 downto 4)+a;

m(6 downto 0):=m(7 downto 1);

else

m(6 downto 0):=m(7 downto 1);

end if;

else

if m(0)='1' then

m(7 downto 4):=m(7 downto 4)-a;

m(6 downto 0):=m(7 downto 1);

else

m(6 downto 0):=m(7 downto 1);

end if;

end if;

--m(6 downto 0):=m(7 downto 1);

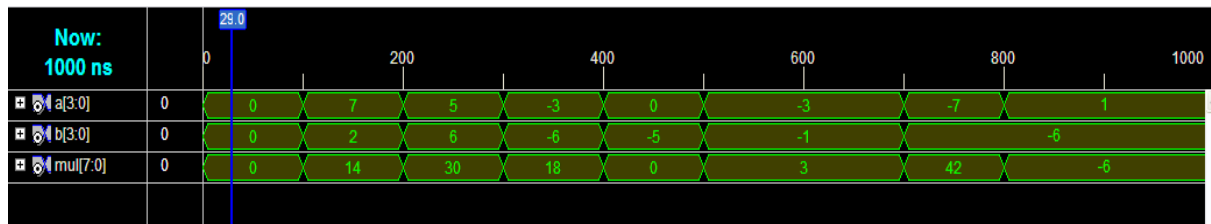
mul<=m;

end process;

end Behavioral;

```

Output for Robertson's Multiplication:



5.2 BOOTH'S MULTIPLICATION:

library IEEE;

```
use IEEE.STD_LOGIC_1164.ALL;
```

```
use IEEE.STD_LOGIC_ARITH.ALL;
```

```
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

entity booths is

Port (a,b : in STD_LOGIC_VECTOR (3 downto 0);

```
mul : out STD_LOGIC_VECTOR (7 downto 0));
```

end booths;

architecture Behavioral of booths is

begin

```
process(a,b)
```

```
variable m: std_logic_vector (7 downto 0);
```

```

variable s: std_logic;

begin

m:="0000"&b;

s:='0';

for i in 0 to 3 loop

if(m(0)='0' and s='1') then

m(7 downto 4):= m(7 downto 4)+a;

s:=m(0);

m(6 downto 0):=m(7 downto 1);

elsif(m(0)='1' and s='0') then

m(7 downto 4):= m(7 downto 4)-a;

s:=m(0);

m(6 downto 0):=m(7 downto 1);

else

s:=m(0);

m(6 downto 0):=m(7 downto 1);

end if;

end loop;

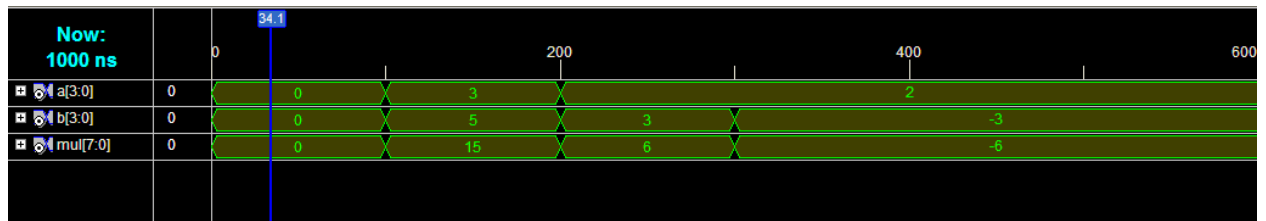
mul<=m;

end process;

end Behavioral;

```

Output for Booth's Multiplication:



5.3 BOOTH'S ORDERING:

library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.STD_LOGIC_ARITH.ALL;

use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity boothorder is

Port (a,b : in STD_LOGIC_VECTOR (7 downto 0);

mul : out STD_LOGIC_VECTOR (14 downto 0));

end boothorder;

architecture Behavioral of boothorder is

begin

process(a,b)

variable m: std_logic_vector(15 downto 0);

```

variable k,l: std_logic_vector(8 downto 0);

begin

m:="0000000000000000";

k(0):='0';

k(8 downto 1):= b(7 downto 0);

l(7 downto 0):=a(7 downto 0);

if(l(7)='0') then

l(8):='0';

else

l(8):='1';

end if;

for i in 0 to 3 loop

if(k(2 downto 0)="000" or k(2 downto 0)="111") then

    m(14 downto 0):= m(15 downto 1);

elsif (k(2 downto 0)="001" or k(2 downto 0)="010") then

    m(15 downto 7):= m(15 downto 7)+1;

    m(14 downto 0):= m(15 downto 1);

elsif (k(2 downto 0)="101" or k(2 downto 0)="110") then

    m(15 downto 7):= m(15 downto 7)-1;

    m(14 downto 0):= m(15 downto 1);

elsif (k(2 downto 0)="011") then

    m(14 downto 0):= m(15 downto 1);

    m(15 downto 7):= m(15 downto 7)+1;

```



```

elsif (k(2 downto 0)="100") then

    m(14 downto 0):= m(15 downto 1);

    m(15 downto 7):= m(15 downto 7)-1;

else

    null;

end if;

if(i<3) then

    m(14 downto 0):= m(15 downto 1);

end if;

k(7 downto 0):= k(8 downto 1);

k(7 downto 0):= k(8 downto 1);

end loop;

mul<=m(14 downto 0);

end process;

end Behavioral;

```

Output For Booth's Ordering:

	200	400	600	800				
X	117	63	-81	-66	-6	100	32	
X	-39	89	121	-3	-33	-34	-38	-54
X	-4563	10413	7623	243	2178	204	-3800	-1728

5.4 PIPELINED QUADRATIC POLYNOMIAL:

```
library ieee;

use ieee.std_logic_1164.all;

use ieee.std_logic_arith.all;

use ieee.std_logic_signed.all;

entity quadratic_func is

generic ( fw : integer := 6 ); -- width of fraction in range 0 to 8

port (

    -- system clock

    clk      : in  std_logic;

    -- clock enable

    en       : in  std_logic;

    -- Coefficients as 8-bit signed fraction

    coeff_a  : in  std_logic_vector(7 downto 0);

    coeff_b  : in  std_logic_vector(7 downto 0);

    coeff_c  : in  std_logic_vector(7 downto 0);

    -- Input as a 8-bit signed fraction

    x_in     : in  std_logic_vector(7 downto 0);

    -- Output as a 24-bit signed fraction
```

```
y_out : out std_logic_vector(23 downto 0));
```

```
end entity;
```

architecture rtl of quadratic_func is

```
signal zeros      : std_logic_vector(23 downto 0);
```

```
signal coeff_a_reg : std_logic_vector(7 downto 0);
```

```
signal coeff_b_reg : std_logic_vector(7 downto 0);
```

```
signal coeff_c_reg : std_logic_vector(7 downto 0);
```

```
signal coeff_c_del : std_logic_vector(7 downto 0);
```

```
signal x2          : std_logic_vector(15 downto 0);
```

```
signal x2_a        : std_logic_vector(23 downto 0);
```

```
signal x2_a_norm    : std_logic_vector(23 downto 0);
```

```
signal x1          : std_logic_vector(7 downto 0);
```

```
signal x1_del       : std_logic_vector(7 downto 0);
```

```
signal x1_b         : std_logic_vector(15 downto 0);
```

```
signal x1_b_norm    : std_logic_vector(15 + fw downto 0);
```

```
signal x0_c_norm    : std_logic_vector(7 + fw*2 downto 0);
```

```
signal sum          : std_logic_vector(23 downto 0);
```

```
signal sum_reg      : std_logic_vector(23 downto 0);
```

```
begin
```

```
-- For padding --
```

```
zeros <= (others => '0');
```

```
-- Rename input x term to maintain naming convention --
```

```

x1 <= x_in;

-- Pipeline the coefficients --

coeff_regs: process (clk)

begin

    if clk'event and clk = '1' then

        if en = '1' then

            coeff_a_reg <= coeff_a;

            coeff_b_reg <= coeff_b;

            coeff_c_reg <= coeff_c;

        end if;

    end if;

end process coeff_regs;

-- Delays to compensate for latencies --

pipe_reg_del: process (clk)

begin

    if clk'event and clk = '1' then

        if en = '1' then

            -- x term requires 1 cycle of delay

            x1_del <= x1;

            -- coeff c requires 1 cycle of delay

            coeff_c_del <= coeff_c_reg;

        end if;

    end if;

end process pipe_reg_del;

```

```

end process pipe_reg_del;

-- x^2 term --

pipe_reg_x2: process (clk)
begin

    if clk'event and clk = '1' then

        if en = '1' then

            x2 <= x1 * x1; -- 8*8 = 16-bits

        end if;

    end if;

end process pipe_reg_x2;

-- x^2 * coeff_a --

pipe_reg_x2_a: process (clk)
begin

    if clk'event and clk = '1' then

        if en = '1' then

            x2_a <= x2 * coeff_a_reg; -- 16*8 = 24-bits

        end if;

    end if;

end process pipe_reg_x2_a;

-- x * coeff_b --

pipe_reg_x1_b: process (clk)
begin

    if clk'event and clk = '1' then

```

```

    if en = '1' then

        x1_b <= x1_del * coeff_b_reg; -- 8*8 = 16-bits

    end if;

end if;

end process pipe_reg_x1_b;

-----

-- 24-bits + 16-bits + 8-bits          --

-- Need to normalize the x1 and c terms so --

-- that the binary points line up      --

-- x1 term << fw, c term << fw*2      --

-----

x2_a_norm <= x2_a;

x1_b_norm <= x1_b    & zeros(fw - 1 downto 0);

x0_c_norm <= coeff_c_del & zeros(fw*2 - 1 downto 0);

-----

-- (x^2 * coeff_a) + (x * coeff_b) + coeff_c (24-bit add) --

-----

sum <= x2_a_norm + x1_b_norm + x0_c_norm;

-----

-- Register the output sum --

-----

```

```

out_reg: process (clk)

begin

    if clk'event and clk = '1' then

        if en = '1' then

            sum_reg <= sum;

        end if;

    end if;

end process out_reg;

-----

-- 24-bit output                --

-- Integer part of result is y_out >> fw*3 --

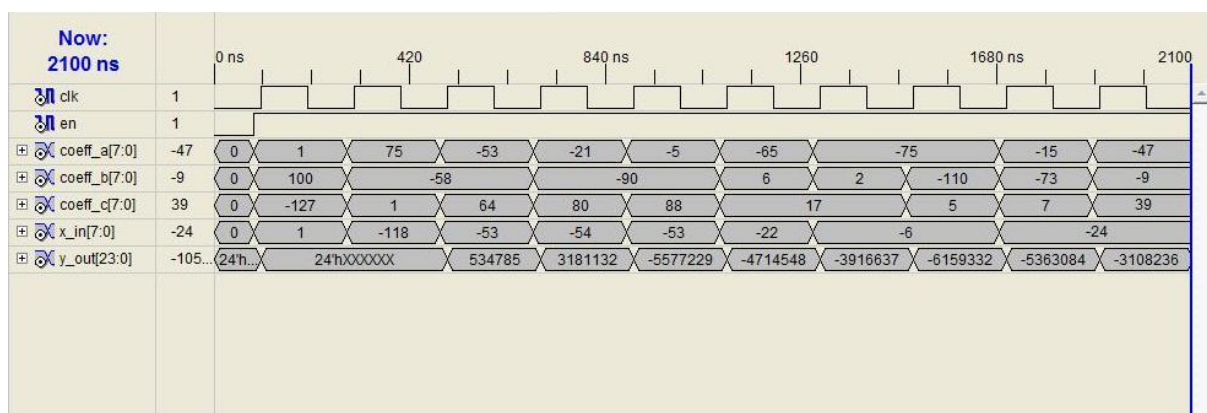
-----

y_out <= sum_reg;

end rtl;

```

Output for Pipelined Quadratic Polynomial:



5.5 CONCLUSION:

From the above output values we can conclude that the pipelined quadratic polynomial has 3 clock cycle latency.

$y = 0.86x^2 - 0.22x + 0.3$ has calculated by varying x value and the fraction width was set to 6 bits. This fraction width we can change however. If we consider x as 0.5 then we are getting $y = 0.405$.

Synthesis report for the selected device: 3s500efg320-5

Number of Slices: 19

Number of Slice Flip Flops: 32

Number of 4 input LUTs: 35

Number used as logic: 27

Number used as Shift registers: 8

Minimum period: 5.613ns (Maximum Frequency: 178.148MHz)

Chapter -6

CONCLUSIONS AND FUTURE WORK

6.1 CONCLUSIONS:

All the Algorithms have been implemented using VHDL in XILINX ISE Simulator.

Booth's algorithm results in reduction in the number of add/subtract steps needed (as compared to the Robertson's algorithm) if the multiplier contains sequences of 1s or 0s.

The worst case scenario occurs in booth's algorithm if $X = 010101::01$, where there are $n/2$ isolated 1s, which forces $n/2$ subtractions and $n/2$ additions. So we go for modified booth's algorithm.

We have presented a modification to the Booth's multiplication algorithm known as Booth's ordering. This modification produces correct results for higher radix fixed point multiplication when the radix is a power of 2.

In this project implementation pipelined of quadratic polynomial function is also described.

6.2 FUTURE WORK:

As implementation of pipelined quadratic polynomial function is described, next we can go for polynomial multiplication, and we can go for FPGA implementation of an efficient multiplier over finite fields can be done.

REFERENCES

1. J.H.Kim, J.S.Lee and J.D.Cho "A Low Power Booth Multiplier Based on Operand Swapping in Instruction Level" Journal of the Korean Physical Society, Vol. 33, No. , January 1998, pp. S258_S261.
2. John Wiley & Sons - 2004 – "Arithmetic and Logic in Computer Systems" A JOHN WILEY & SONS, INC., PUBLICATION.
3. P. E. Madrid, B. Millar, and E. E. Swartzlander, "Modified Booth algorithm for high radix fixed-point multiplication," *IEEE Trans. VLSI Syst.*, vol. 1 , no. 2, **pp.** 164-167, June 1993.
4. Rajendra Katti "A Modified Booth Algorithm for High Radix Fixed-point Multiplication" *IEEE Trans. VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS*, VOL 2. NO 1. Dec. 1994.
5. J. J. F. Cavanagh, Digital Computer Arithmetic, New York: McGraw Hill, 1984.
6. L. Song, K.K. Parhi, "Efficient Finite Field Serial/Parallel Multiplication", Proc. of International Conf. On Application Specific Systems, Architectures and Processors, pp. 72-82, Chicago,USA, 1996.
7. J.Lopez and R. Dahab. "Fast Multiplication on Elliptic Curves over $GF(2^m)$ without Precomputation". In C.K. Koc and C. Paar (Eds.): Cryptography Hardware and Embedded Systems, CHES 1999, LNCS, Springer- Verlag, pp. 316-327, 1999.
8. M.A.G. Martinez, R.P. Gomez, G.M. Luna and F.R. Henrique, "FPGA Implementation of an Efficient Multiplier over Finite Fields $GF(2^m)$ " Proceedings of International Conf. On Reconfigurable Computing and FPGAs, 2005.
9. Stuart F. Oberman, and Michael J. Flynn, "Division Algorithms and Implementations" IEEE Transactions on Computers, vol. 46, no. 8, August 1997.
10. www.xilinx.com